DOCUMENT RESUME

ED 396 680 IR 017 856

AUTHOR Holt, Richard C.
TITLE Object-Oriented Programming in High Schools the Turing Way.
PUB DATE 94
NOTE 9p.; In: Recreating the Revolution. Proceedings of the Annual National Educational Computing Conference (15th, Boston, Massachusetts, June 13-15, 1994); see IR 017 841.
PUB TYPE Guides - Classroom Use - Teaching Guides (For Teacher) (052) -- Reports - Descriptive (141) -- Speeches/Conference Papers (150)

EDRS PRICE MF01/PC01 Plus Postage.
DESCRIPTORS *Computer Science Education; *Educational Objectives; Foreign Countries; High Schools; Instructional Materials; *Learning Modules; Programming; *Programming Languages
IDENTIFIERS *Object Oriented Programming

ABSTRACT

This paper proposes an approach to introducing object-oriented concepts to high school computer science students using the Object-Oriented Turing (OOT) language. Students can learn about basic object-oriented (OO) principles such as classes and inheritance by using and expanding a collection of classes that draw pictures like circles and happy faces. Materials are outlined for a two-week teaching unit which support this approach. The units cover: (1) three foundational OO concepts: objects, classes and inheritance; (2) diagrams and relations; (3) software development environments; (4) the OOT language and environment; (5) OOT in an undergraduate curriculum; and (6) OOT at the high school level. (Contains 17 references.) (Author/BEW)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

ED 396 680

U.S. DEPARTMENT OF EDUCATION
Office of Educational Research and Improvement
EDUCATIONAL RESOURCES INFORMATION
CENTER (ERIC)
☐ This document has been reproduced as received from the person or organization originating it.
☐ Minor changes have been made to improve reproduction quality.
• Points of view or opinions stated in this document do not necessarily represent official OERI position or policy.

"PERMISSION TO REPRODUCE THIS MATERIAL HAS BEEN GRANTED BY
Donella Ingham
TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)."

Paper (T4-201A)

Object-Oriented Programming in High Schools the Turing Way

Richard C. Holt
Department of Computer Science
University of Toronto
8 King's College Road
Toronto Canada M5S 1A4
(416) 978-8726
Fax: (416) 978-4765
holt@csri.toronto.edu

Key words: teaching programming, object-oriented programming, computer science education, inheritance, teaching unit, Turing

IR017856

BEST COPY AVAILABLE

Abstract

This paper proposes an approach to introduce object-oriented concepts to high school students using the Object-Oriented Turing language. The students learn about the concepts of objects, classes and inheritance by using and expanding a collection of classes that draw pictures such as circles and happy faces. Materials for a two-week teaching unit have been developed to support this approach.

Introduction

There is a race to develop new software to meet the ever increasing capacity of hardware. Hardware capacity, in terms of both speed and memory, continues to double approximately every two years. Those individuals and countries with the expertise to develop such software have the potential to realize great technological and economic gains.

This continuing demand for software requires new methods of development that help solve the central problem of software creation. The basis of the problem lies in the inherently complex nature of software. What is needed are methods that help us better understand and control the software being developed. The advent of structured programming some years ago was a large step in the direction of controlling this complexity. Object oriented (OO) programming now promises similar gains, because it divides software into distinct parts, called "objects" which communicate only in rigidly specified ways.

After reviewing the key principles of object oriented programming that our students should learn, this paper discusses the Object-Oriented Turing system, which was developed to support the teaching of programming and software engineering. Next comes a discussion of the use of this system in teaching OO ideas across a university curriculum. Finally, the paper presents experience using of the system with high school students and suggests how the system may be useful in high school Computer Science courses.

Principles of OO that We Should Teach

In this section, we will cover the key OO principles [Cox 87, Booch 91, Budd 91, Meyer 88] that a student should learn. It should be emphasized that the ideas of object orientation reach well beyond computer programming. Indeed, we should think of the OO approach as a method of problem solving [Yoder 1993], which applies well to programming.

Three Foundational OO Concepts: Objects, Classes and Inheritance

Objects

The first and most important concept in OO is called information hiding. Long before OO became popular, this concept was recognized as the "black box" principle. A black box is an item, such as a radio, that is understood in terms of its inputs and outputs and not by its internal construction. For the radio, its buttons and knobs characterize its input and the sound it produces characterizes its output. In OO terminology we refer to a black box as an "object". In terms of software, the essence of an object or black box is that we hide data (and other internal implementation details) in a "box" and all that we can manipulate or observe from the outside is the externally visible interface, mainly the exported subprograms (these are called *methods*). OO languages provide syntactic mechanisms to enforce this hiding.

Classes

The second foundational concept in OO is the idea of a "class". A "class" of objects is a set of objects all of which are the same, or sufficiently the same for our purposes. From a programming point of view, a class is a template from which we can *instantiate* or replicate objects. Using our previous example, a class can be thought of as the design of a radio, from which we can make many individual radios. In large software systems, we make constant use of software objects, such as files and windows, which are (or are essentially) instances of classes. To make the concept of classes clear to the student, we need exercises that use many objects. The book *An Introduction to Object-Oriented Programming* [Budd 91] gives a good example of the use of objects, namely, a program that supports the game of solitaire using a class of playing cards and a class of stacks of card .

Inheritance

The third key OO concept is called "inheritance". Class D *inherits* from class C if class D contains all the items of interest that C contains. For example, consider a new radio design D, that is just like an old design C, except D adds a new knob that activates a new external plug for ear phones. In this case, we say D inherits from C. We say D *is a* C, meaning that we can use a radio of design D for all the same purposes for which we can use a radio of design C.

There are actually two ways in which an inheriting class D can be different from its parent class C. First, it can extend or add to the parent class. For example, a new subprogram or data field can be added. Second, it can change or *override* certain kinds of items in the parent. The power button on the radio might be changed (overridden) to turn on a light on the radio, as well as turning on the internal circuits.

Overriding allows us to create similar but significantly different objects. For example, all objects that are Macintosh files have much in common, but the effect of opening a particular file depends on the kind of file we open. For example, opening a *HyperCard* file is quite different from opening a Word Perfect file. This is because the open operation has been effectively overridden for the various kinds of files. We say that Macintosh files are *polymorphic* because they react in varying ways to the same operations.

These are the three basic concepts of OO (objects, classes and inheritance). In this discussion we have used the metaphor of a radio and its design. We now turn to diagrammatic conventions for representing OO concepts.

Diagrams and Relations

There are a number of important relationships among objects and classes, and these are best understood using diagrammatic conventions. At the level of objects (instances), the two key relations are *has a* and *uses*. Figure 1 gives a diagram of an Account Manager object and a Check Book object. The arrow from the Account Manager to the Check Book indicates that the Account Manager "uses" (calls) the Check Book. The Check Book object "has a" (contains) internal variables (a ledger, which is an array of records that keep track of checks and withdrawals) and two externally visible subprograms (Write Check and Make Withdrawal). The protrusion of Write Check and Make Withdrawal from Check Book indicates that they are visible outside of Check Book. To keep the diagram simple, we do not show the items contained in the Account Manager.

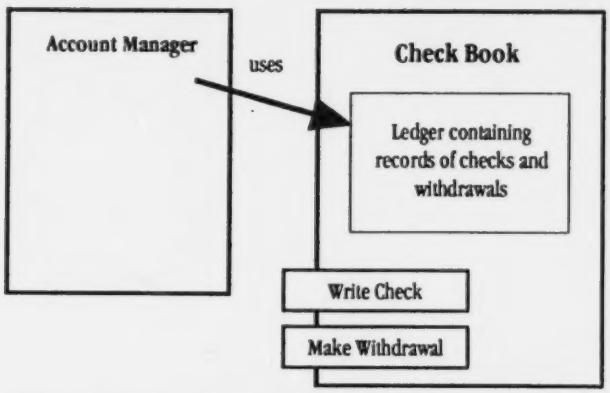

Figure 1. Example of diagrammatic conventions for objects, showing the "uses" relation and the "has a" relation.

Among classes, the most important relation is "inherits", as illustrated in Figure 2, which shows that radio design D inherits from radio design C. Another essential relation is "instance of". For example, a particular radio object R, might be an instance of radio class D.

BEST COPY AVAILABLE

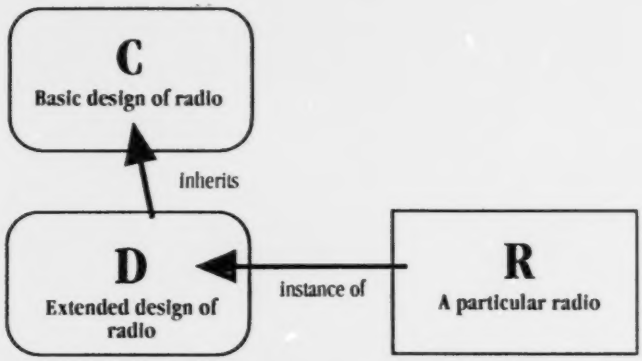

Figure 2. Diagrammatic convention for classes and objects, showing the "Inherits" and "instance of" relations. Rectangular boxes are objects. Rounded boxes are classes.

Students will understand OO ideas much better when they have mastered these diagrammatic techniques. (There are many diagrammatic conventions; the details of the one used for teaching are not important.) A student should be able to visualize a given program's structure in terms of these diagrams and conversely, be able to create a program that has been designed using these diagrams.

The reason these diagrams are so useful for learning, is that they use our visual sensibilities to represent a rich set of ideas. These ideas include the relations of "has a", "uses", "is a", and "instance of" among objects and classes. These diagrams concentrate on the software's structure, allowing us to suppress implementation detail to better understand program design.

Software Development Environments

One of the most important ideas emerging from the Smalltalk language is that a programming environment, based on appropriate principles, can significantly improve the way we program. A Smalltalk environment includes on-line libraries. We can *browse* through these libraries of re-usable classes and experiment with them with great facility. Largely because of this, Smalltalk encourages rapid prototyping to a degree that has not been approached in most languages. Turbo (Borland) environments for Pascal and C++ provide integrated environments that expedite the edit-compile-link-debug cycle. Compared with those environments, the Smalltalk environment has the advantage of supporting larger scale programming, in which off-the-shelf components can be assembled into new programs.

Missing from both Smalltalk and Turbo environments are tools that are commonly called CASE (Computer Aided Software Engineering) facilities. These tools provide machine assistance for the diagrammatic approach that we have just discussed. The student's learning can be greatly advanced if these tools are available in an integrated environment, which ideally can generate diagrams from software, can generate skeletal software from diagrams, and can check that the diagrams and software are consistent. The OOT environment, which is described below, provides both diagrammatic capability and on-line re-use libraries.

Software development environments (SDEs), as exemplified by Smalltalk, support a method of software creation that is inherently faster and better than is possible using older paradigms. Those paradigms were limited to collections of tools such as editors, compilers, linkers, and debuggers.

It is clear that the fundamental concepts of OO are closely related to ideas such as diagrammatic conventions, software development environments and software re-use. We will now discuss OOT, which is a language and environment suitable for teaching OO ideas.

The OOT Language and Environment

The C++ and Smalltalk languages are perhaps the most commonly mentioned languages for supporting the teaching of OO concepts. Other languages which are good candidates for teaching OO concepts include Objective C, Turbo Pascal with OO extensions, Eiffel, Modula 3, and CLOS. We will not discuss these languages in any detail, but refer the reader to discussions of OO languages appearing in the literature [Budd 91, Booch 91]. In this section we will give an overview of the Object-Oriented Turing language and software development system.

The OOT software development environment was designed for use in teaching. It has evolved from the Turing language [Holt 88] which is a Pascal-like language that is very easy to learn. Turing is now used in 30 universities and in half of the high schools (about 400 schools) in the Province of Ontario, where it is used on PCs and Macintoshes. The Turing implementation used in high schools provides an integrated edit-compile-run system. OOT extends the basic Turing system by providing more advanced programming features, including OO features, and a sophisticated software development environment. The OOT environment, up to now, has been used on Unix systems including SUNs, SGIs and IBM RS-6000's. A version that runs on PCs under MS-Windows is expected to be available in Summer 1994. The OOT language and its SDE have been described elsewhere [Mancoridis 92, Holt 92], so we shall only give an overview here. (See the Appendix for the way to access the FTP on-line Unix demonstration of OOT.)

With the advent of windowing systems, such as MicroSoft Windows for PCs, our students should be aware of the ways in which windowing facilitates programming. In the case of the OOT SDE, individual windows encapsulate the distinct ideas that the programmer deals with. For example, each source program object, such as the Account Manager in Figure 1, is displayed in its own window. There are also windows to show the program's output and to show error messages.

Perhaps OOT's most striking use of windows is the Landscape Window. This window displays a diagram (called the Landscape) with boxes representing the objects and classes in the program. For example, a Landscape view might show a picture much like Figure 1 for the corresponding program. There is a "hot link" that allows the user to immediately access the source code that corresponds to each box in the diagram.

Another window gives the Process Dump, which is a stack trace of called procedures that can be used to locate the current line of execution. There are also windows, called Interface Views, that give the interfaces for objects such as the Account Manager object (Figure 1). These Interface Views, which are automatically created from the source programs, allow the programmer to inspect the entry points (methods) of an object along with corresponding parameter types and comments. Double clicking on the name of a method in an Interface View causes the corresponding source code to pop up in a window.

OOT displays the current directory in a window, much as does a Macintosh. Double clicking on a name in this Viewer causes a fresh window to be popped up. This new window shows the file's contents. The Directory Viewer serves as a browser for inspecting objects and classes, as well as for the on-line language reference manual and for re-use libraries.

Our purpose here is not to describe OOT in any detail [Holt 92] but rather clarify how much its facilities may be of help in teaching programming and Computer Science concepts.

OOT in an Undergraduate Curriculum

This section discusses the use of OOT [Holt 1993] to introduce object-oriented concepts across the undergraduate curriculum [Temte 91, Reid 92]. The following section will focus on use of OOT with high school students.

At the University of Toronto, OOT is used in many courses, including courses on data structures, courses on compilers and courses on operating systems. We will concentrate here on those courses which have used OOT explicitly for teaching object-oriented concepts. In particular, we will discuss our use of OOT for teaching these concepts in (1) an introductory programming course, (2) a course on programming paradigms and (3) a course on software engineering.

Since Fall 1992, the OOT software development environment has been used in the University of Toronto introductory classes in the Faculty of Applied Science and Engineering. These classes have followed a fairly traditional approach to introductory computing at the university level, with emphasis on general computing concepts such as data structuring, operating systems and networks as well as programming proper. For the first time, in Fall 1993, a unit in one of these classes is concentrating on object-orientation, with the goal of making students aware of the concepts of objects, classes and inheritance. The students are required to complete a graphics-based assignment (much like the one described below for the use in high schools).

When these students are first introduced to OOT, they use only simple features, including a window containing their program, an output window and the Directory Viewer.

OOT has been designed to be very easy for the novice to use [Milbrandt 1991]. The student begins, with little explicit instruction, by using OOT's mouse-based Macintosh-like interface. The Turing language's simple input/output and graphics statements allow students to begin writing programs immediately. For example, here is a complete program that outputs "Hello world" and draws a green box on the screen with opposite corner coordinates at (10, 15) and (100, 120).

```
put "Hello world"
drawbox (10, 15, 100, 120, Green)
```


BEST COPY AVAILABLE

The OOT system has a pay-as-you-play philosophy that allows students to learn more of the system as they master more concepts. Only after the fundamentals of programming, including loops, arrays and subprograms, have been taught, are OO concepts introduced.

Like many Computer Science departments, ours offers a course on programming paradigms, which is given in the third year. Our course is actually titled Principles of Programming Languages, but its real purpose is to acquaint students with paradigms such as logic programming (PROLOG), functional programming (LISP), concurrency, and so on.

In a 4-week unit in this course, we use OOT to teach the OO paradigm. This unit has as its goal to teach design implications of OO, at a much deeper level than is possible in an introductory course. By the time our students reach this course, they have had considerable experience with Turing, though not generally its OO features, so little time is wasted instructing them about syntactic issues of OOT. Although other languages, such as C++, are mentioned in the unit, only OOT is covered in any depth. The students' assignments are based on an existing class-intensive program called Star, which reads OOT programs and automatically creates diagrams on the screen for them that are analogous to Figure 2. The students are required to enhance this program in various ways, for example, so it outputs PROLOG facts corresponding to the relations among the OOT program's classes. This work exposes students to many issues of importance to software engineering, including automatic program diagramming, program maintenance, and multiple views of a program, all within an OO context.

A fourth year University of Toronto course provides a standard coverage of software engineering concepts [Sommerville 92]. This course uses an OO approach in the following way. At the beginning of the course, the students are given a set of milestones representing the phases in the life cycle of a software "product" that teams of three students in the course are required to create. The product last year was a Graphical User Interface (GUI) library written in OOT targeted for use by other undergraduates. Ideally, the product would be used by other classes of students to allow them to incorporate GUI support (menus, buttons, etc.) as a part of their programs.

The hardest part of the project was the design phase, namely, deciding upon the class hierarchy for user interface objects. The goal was to provide an interesting exercise in software design, something that is too often missing in undergraduate education. This project was intended to teach many software engineering concepts including the software life cycle, team work, delivery of re-usable software, etc., all in a modern OO environment, namely OOT.

This short discussion of use of OOT at the University of Toronto has been intended to illustrate how the teaching of OO concepts is being introduced at the university level. We now turn to the question of teaching these concepts in high schools [McGregor 1992, Stephenson 1992, Funkhouser 1993].

OOT at the High School Level

Each summer the Department of Computer Science at the University of Toronto teaches a short summer course to selected high school students on principles of Computer Science. Last summer 62 students participated in this intensive 3-week course.

For the last two summers, these students have been introduced to OO concepts, in a unit in this summer course, in the following way. First the students practice the fundamentals of programming: loops, IFs, subprograms and simple data structures, as well as simple graphics. Then, they are exposed to OO concepts (objects, classes, and inheritance) by means of an assignment based on a simple class library for drawing figures. This library (see Figure 3) consists of a tree of classes.

BEST COPY AVAILABLE

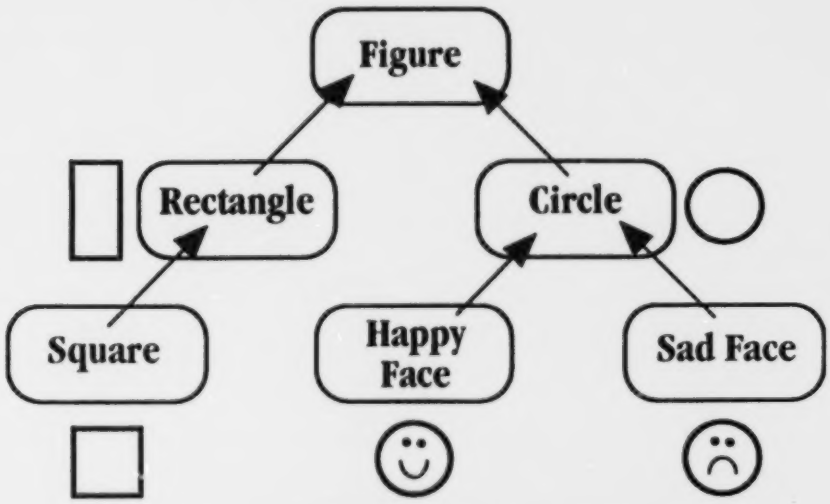

Figure 3. A class hierarchy used for introducing students to OO concepts. The arrows show inheritance.

The root of the class tree is called Figure. It represents objects that can, in principle, be drawn on the screen and later erased. The actual bodies for the draw and erase procedures of the Figure class are omitted. In other words, Figure is an "abstract" (or "virtual") class that represents all objects that can be drawn and erased, but does not represent any particular figure. Classes such as Rectangle and Circle, which descend from (inherit from) the Figure class, provide actual code to draw and erase particular figures. (In the actual assignment, there is also code to set the color, size and position of figures, but these details will be ignored in this paper.)

The students are provided with a library that implements the hierarchy shown in Figure 3, but without the HappyFace and SadFace classes. The students are required to enhance the library with the missing classes and to have their program draw an interesting scene on the screen using these classes.

Since the assignment is so graphical, it is easily explained to students. They are clearly pleased with the results of their work, which displays a picture on the screen. The objective of this exercise is to give introductory students a good feel for OO concepts, including use of libraries and SDEs.

This experience with high school students suggests that teaching object-orientation at the high school level, using software support such as that provided by OOT, is straightforward and can be quite rewarding to the students. The approach assumes an initial introduction to fundamental programming ideas and follows immediately with OO ideas. The emphasis on graphics makes the work exciting to the students and makes the ideas much easier to grasp.

Using the experience from these courses, the author has developed a two-week OO teaching unit for use in high schools. The unit is supported by a 19 page hand out for the students, which includes exercises. Each concept is first introduced at the "everyday level", that is, in terms of familiar objects such as radios. Then the students are introduced to the corresponding OO concept at the level of programming. The students use a collection of classes, similar to the collection described here, to gain experience with actual OO programming, using the Object-Oriented Turing system. The student should have experience with a programming language such as Pascal, C or Basic and a familiarity with procedures before covering this unit. The Appendix tells how to access this teaching material and software.

The state of the art in computer software is constantly changing [Stephenson 1990]. We should not be surprised that there are new ideas, such as OO concepts, that need to be introduced into our schools. The design of Computer Science curricula for high schools [Merrit 1993] is a never ending job, repeatedly introducing new concepts when they are seen to be intellectually interesting and industrially justified. It seems clear that OO concepts have now reached this stage, and it is only a question of how, not whether to introduce these ideas into our schools. The good news is that OO concepts can be nicely integrated with high school Computer Science teaching without a great deal of change in the approaches we have already been using.

BEST COPY AVAILABLE

Conclusions

This paper suggests that, given an appropriate software development environment such as OOT, sophisticated OO ideas can and should be taught at the high school level. A radical change is not required in teaching programming fundamentals. Instead, once the fundamentals are introduced, a new direction, the OO direction, is followed in a natural and rewarding way.

References

[Budd 1991] Budd, T. *An Introduction to Object-Oriented Programming*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1991.

[Booch 1991] Booch, G. *Object-Oriented Design with Applications*. Benjamin/Cummings, Redwood City, Calif, 1991.

[Cox 1987] *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley Publishing Company, Reading, Massachusetts.

[Funkhouser 1993] Funkhouser, C. OOPS? Its Basic, Journal of Computer Science Education, Summer 1993, pp. 21-27.

[Holt 1988] Holt, R. C. and Cordy, J.R. The Turing Programming Language. Comm. ACM 31, 12 (Dec. 1988), 1410-1423.

[Holt 1992] Holt, R. C. *Turing Reference Manual*, Third Edition, Holt Software Associates Inc., March 1990, 361 pages, Toronto.

[Holt 1993] Holt, R. C. Introducing Undergraduates to Object Orientation Using the Turing Language, Department of Computer Science, University of Toronto, July 1993 (unpublished).

[Mancoridis 1993] Mancoridis, S., Holt, R., and Penny, D. A "Curriculum-Cycle" Environment for Teaching Programming. 24th SIGCSE Technical Symposium, Assoc. for Computing Machinery, Feb. 18-19, Indianapolis, Indiana, SIGCSE Bulletin 25, 1 (Mar. 1993).

[McGregor 1992] The Role of Object-Oriented Development Techniques in Computer Science Education. John McGregor, Moderator, Proceedings NECC 92, Dallas, pg. 40.

[Merrit 1993] Merrit, S. ACM Model High School Curriculum, Session in Proceeding of NECC 93, Orando, June 1993.

[Meyer 1988] Meyer, B. *Object-Oriented Software Construction*. Prentice-Hall International, London, 1988.

[Milbrandt 1991] Milbrandt, G. Comparison of BASIC, Turing, Pascal and C for Computer Studies Courses, Journal of Computer Science Education, Summer 1991, pp. 11-14.

[Stephenson 1990] Stephenson, C. Changing Trends in High School Programming, Journal of Computer Science Education, Winter 1990, pp. 6-11.

[Reid 1992] Reid, R. The Object-Oriented Paradigm in CS1. 24th SIGCSE Technical Symposium, Assoc. for Computing Machinery, Feb. 18-19, Indianapolis, Indiana, SIGCSE Bulletin 25, 1 (Mar. 1993), pp. 265-269.

[Sommerville 1992] Sommerville, Ian, *Software Engineering*, Fourth Edition. Addison-Wesley, 649 pp., 1992.

[Temte 91] Temte, M.c2. Let's Begin Introducing the Object-Oriented Paradigm. SIGSCE Bulletin 23, 1 (March 1991), 73-77.

[Yoder 1993] Yoder, S. and Moursund, D. Do Teachers Need to Know About Programming? Journal of Computing in Teacher Education, Vol. 9, No. 3, Spring 1993, pp. 21-26.

Appendix: Access to Teaching Unit and OOT Software

A copy of the two-week unit for teaching OO concepts is available from the author. A demonstration version of the Object-Oriented Turing system for MicroSoft Windows can also be requested. This version comes with the collection of classes used in the teaching unit.

For those people with access to Unix, there is an on-line demonstration version of Unix OOT from the University of Toronto that can be accessed by anonymous FTP (File Transfer Protocol). The OOT environment has been implemented on various Unix platforms, such as Sun/4's, RS/6000 and SGI. If you have access to the Internet and Unix, you can get instructions to access the demo by these commands on Unix:

```
%ftp 128.100.1.192
ftp> cd pub
ftp> get ootDistrib
ftp> quit
```

The ootDistrib file in your directory will now contain details on getting the demo.